FOUNDATION-0.7: Agent Communication & Task Routing - PART 2 (Execution & Testing)

🎯 CONTEXT

Phase: FOUNDATION (Week 1 - Day 3 Afternoon)

Component: Agent Communication & Task Routing - Execution & Validation

Estimated Time: 15 min execution + 20 min testing

Complexity: MEDIUM-HIGH

Risk Level: LOW

Files: Part 2 of 2 (Execution, testing, validation)

📦 PREREQUISITES

Must Have Completed:

- ✅ **PART 1** - All code files created
- ✅ **FOUNDATION-0.6** - Agent Registry
- ✅ **P-02** - Orchestrator skeleton
- ✅ Redis running and healthy

Verify PART 1 Files Exist:

```bash
cd ~/optiinfra

# Check Go files
ls -lh services/orchestrator/internal/task/models.go
ls -lh services/orchestrator/internal/task/router.go
ls -lh services/orchestrator/internal/task/handlers.go

# Check Python files
ls -lh shared/orchestrator/task_handler.py
ls -lh agents/cost_agent/main.py

# Expected: All files present
```

🚀 STEP-BY-STEP EXECUTION

Step 1: Create Go Directory Structure

bash

```
cd ~/optiinfra/services/orchestrator

# Create task package directory
mkdir -p internal/task

# Verify structure
tree -L 3 internal/
# Expected:
# internal/
# ├── registry/
# │   ├── handlers.go
# │   ├── models.go
# │   └── registry.go
# └── task/          <- NEW
```

Validation:

bash

```
# Check directories exist
[ -d "internal/task" ] && echo "✅ internal/task/ created" || echo "❌ Missing"
```

Step 2: Create Task Models

bash

```
cd ~/optiinfra/services/orchestrator/internal/task

# Create models.go (copy from PART 1, FILE 1)
cat > models.go << 'EOF'
package task

import (
  "time"
)
[... COPY ENTIRE MODELS.GO FROM PART 1, FILE 1 ...]
EOF

# Verify file
ls -lh models.go
wc -l models.go
# Expected: ~200 lines
```

Validation:

```
bash

# Check file size
FILE_SIZE=$(wc -l < models.go)
if [ "$FILE_SIZE" -gt 150 ]; then
    echo "✅ models.go created ($FILE_SIZE lines)"
else
    echo "❌ models.go too small ($FILE_SIZE lines)"
fi

# Check for key types
grep -q "type Task struct" models.go && echo "✅ Task struct found" || echo "❌ Missing"
grep -q "type TaskType" models.go && echo "✅ TaskType found" || echo "❌ Missing"
grep -q "type TaskStatus" models.go && echo "✅ TaskStatus found" || echo "❌ Missing"
```

Step 3: Create Task Router

```
bash
```

```
cd ~/optiinfra/services/orchestrator/internal/task

# Create router.go (copy from PART 1, FILE 2)
cat > router.go << 'EOF'
package task
[... COPY ENTIRE ROUTER.GO FROM PART 1, FILE 2 ...]
EOF

# Verify file
ls -lh router.go
wc -l router.go
# Expected: ~450 lines
```

Validation:

```
bash

# Check file size
FILE_SIZE=$(wc -l < router.go)
if [ "$FILE_SIZE" -gt 400 ]; then
    echo "✅ router.go created ($FILE_SIZE lines)"
else
    echo "❌ router.go too small"
fi

# Check for key functions
grep -q "func NewRouter" router.go && echo "✅ NewRouter found" || echo "❌ Missing"
grep -q "func.*SubmitTask" router.go && echo "✅ SubmitTask found" || echo "❌ Missing"
grep -q "func.*executeTask" router.go && echo "✅ executeTask found" || echo "❌ Missing"
```

Step 4: Create HTTP Handlers

```
bash
```

```
cd ~/optiinfra/services/orchestrator/internal/task

# Create handlers.go (copy from PART 1, FILE 3)
cat > handlers.go << 'EOF'
package task
[... COPY ENTIRE HANDLERS.GO FROM PART 1, FILE 3 ...]
EOF

# Verify file
ls -lh handlers.go
wc -l handlers.go
# Expected: ~100 lines
```

Step 5: Update Main Server

bash

```
cd ~/optiinfra/services/orchestrator/cmd/server

# Update main.go (copy from PART 1, FILE 4)
cat > main.go << 'EOF'
package main
[... COPY ENTIRE MAIN.GO FROM PART 1, FILE 4 ...]
EOF

# Verify file
ls -lh main.go
```

Step 6: Update Go Dependencies

bash

```
cd ~/optiinfra/services/orchestrator

# Update go.mod if needed
go mod tidy
go mod download

# Expected: Dependencies up to date
```

Validation:

```bash
# Verify dependencies
go list -m all | head -15

# Check if key dependencies present
go list -m github.com/gin-gonic/gin && echo "✅ gin installed" || echo "❌ Missing"
go list -m github.com/go-redis/redis/v8 && echo "✅ redis installed" || echo "❌ Missing"
go list -m github.com/google/uuid && echo "✅ uuid installed" || echo "❌ Missing"
```

Step 7: Build Updated Orchestrator

```bash
cd ~/optiinfra/services/orchestrator

# Build the Go binary
go build -o bin/orchestrator ./cmd/server

# Verify binary created
ls -lh bin/orchestrator
# Expected: Binary file ~12-25MB

# Check build errors
echo $?
# Expected: 0 (success)
```

Validation:

```bash
# Check if binary was created
if [ -f "bin/orchestrator" ]; then
    echo "✅ Orchestrator binary built successfully"
    ls -lh bin/orchestrator
else
    echo "❌ Build failed - check for errors above"
    exit 1
fi
```

Step 8: Create Python Task Handler

```
bash

cd ~/optiinfra/shared/orchestrator

# Create task_handler.py (copy from PART 1, FILE 5)
cat > task_handler.py << 'EOF'
"""
Task handler for Python agents.
[... COPY ENTIRE TASK_HANDLER.PY FROM PART 1, FILE 5 ...]
"""
EOF

# Verify file
ls -lh task_handler.py
wc -l task_handler.py
# Expected: ~200 lines
```

Step 9: Update Python init.py

```
bash

cd ~/optiinfra/shared/orchestrator

# Update __init__.py (copy from PART 1, FILE 6)
cat > __init__.py << 'EOF'
"""
Orchestrator client utilities for Python agents.
"""

from shared.orchestrator.registration import AgentRegistration
from shared.orchestrator.task_handler import TaskHandler

__all__ = ['AgentRegistration', 'TaskHandler']
EOF

# Verify file
ls -lh __init__.py
```

Step 10: Create Example Cost Agent

```bash
cd ~/optiinfra

# Create agents directory structure
mkdir -p agents/cost_agent

cd agents/cost_agent

# Create main.py (copy from PART 1, FILE 7)
cat > main.py << 'EOF'
#!/usr/bin/env python3
"""
Cost Agent - Example implementation.
[... COPY ENTIRE MAIN.PY FROM PART 1, FILE 7 ...]
"""
EOF

# Make executable
chmod +x main.py

# Verify file
ls -lh main.py
wc -l main.py
# Expected: ~250 lines
```

Step 11: Install Python Dependencies

```bash
cd ~/optiinfra

# Install Flask for task handler
pip install flask requests

# Verify installation
python -c "import flask; import requests; print('✅ Dependencies installed')"
```

Step 12: Stop Old Orchestrator (if running)

```bash
# Find and stop old orchestrator process
pkill -f "orchestrator" || echo "No orchestrator running"

# Verify stopped
sleep 2
ps aux | grep orchestrator | grep -v grep || echo "✅ Orchestrator stopped"
```

Step 13: Start Updated Orchestrator

```bash
cd ~/optiinfra/services/orchestrator

# Set environment variables
export REDIS_ADDR="localhost:6379"
export PORT="8080"

# Run orchestrator
./bin/orchestrator

# Expected output:
# Connected to Redis
# Agent registry started
# Task router initialized
# Starting orchestrator on port 8080
```

Keep this terminal running! Open a new terminal for next steps.

Step 14: Test Health & Registry (New Terminal)

```bash
```

```
# In a NEW terminal
cd ~/optiinfra

# Test orchestrator health
curl http://localhost:8080/health

# Expected output:
# {
#   "service": "orchestrator",
#   "status": "healthy",
#   "timestamp": "2025-10-20T..."
# }

# Check agents endpoint
curl http://localhost:8080/agents

# Expected: {"agents": [], "count": 0}
```

Validation:

```
bash

# Test with validation
RESPONSE=$(curl -s http://localhost:8080/health)
if echo "$RESPONSE" | grep -q "healthy"; then
    echo "✅ Orchestrator is healthy"
else
    echo "❌ Health check failed"
    echo "Response: $RESPONSE"
fi
```

Step 15: Start Cost Agent

```
bash
```

```
# In the SAME new terminal (not the orchestrator terminal)
cd ~/optiinfra/agents/cost_agent

# Set Python path
export PYTHONPATH="$HOME/optiinfra:$PYTHONPATH"

# Run cost agent
python main.py

# Expected output:
# Starting cost-agent-1...
# Task handler started on 0.0.0.0:8001 (threaded)
# Agent registered successfully: abc12345-...
# Heartbeat started
# cost-agent-1 is running and ready to receive tasks
```

Keep this terminal running too! Open a THIRD terminal for testing.

Step 16: Verify Agent Registration

bash

```
# In a THIRD new terminal
cd ~/optiinfra

# Check if agent registered
curl http://localhost:8080/agents

# Expected output:
# {
#   "agents": [
#     {
#       "id": "abc12345-...",
#       "name": "cost-agent-1",
#       "type": "cost",
#       "status": "healthy",
#       "capabilities": ["analyze_cost", "migrate_to_spot", "right_size", ...],
#       ...
#     }
#   ],
#   "count": 1
# }
```

Validation:

bash

```
# Verify agent is registered and healthy
RESPONSE=$(curl -s http://localhost:8080/agents)
if echo "$RESPONSE" | grep -q "cost-agent-1"; then
    echo "✅ Cost agent registered successfully"
else
    echo "❌ Agent not found"
    echo "Response: $RESPONSE"
fi
```

Step 17: Submit Test Task - Cost Analysis

bash

```bash
# Submit a cost analysis task
curl -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "task_type": "analyze_cost",
    "agent_type": "cost",
    "parameters": {
      "account_id": "test-account-123",
      "period": "last_7_days"
    },
    "priority": 5,
    "timeout_seconds": 30
  }'

# Expected output:
# {
#   "task_id": "550e8400-...",
#   "status": "pending",
#   "agent_id": "abc12345-...",
#   "created_at": "2025-10-20T10:30:00Z",
#   "status_url": "/tasks/550e8400-..."
# }
```

Save the task_id from the response!

Step 18: Check Task Status

bash

```
# Replace TASK_ID with actual task ID from step 17
TASK_ID="your-task-id-here"

# Get task status
curl http://localhost:8080/tasks/$TASK_ID

# Expected output (after a few seconds):
# {
#   "task_id": "550e8400-...",
#   "status": "completed",
#   "agent_id": "abc12345-...",
#   "result": {
#     "account_id": "test-account-123",
#     "current_spend": 15420.50,
#     "potential_savings": 6800.25,
#     "savings_percentage": 44.1,
#     "recommendations": [...]
#   },
#   "created_at": "2025-10-20T10:30:00Z",
#   "started_at": "2025-10-20T10:30:01Z",
#   "completed_at": "2025-10-20T10:30:03Z",
#   "retry_count": 0
# }
```

Validation:

bash

```
# Check task status with validation
RESPONSE=$(curl -s http://localhost:8080/tasks/$TASK_ID)
if echo "$RESPONSE" | grep -q "completed"; then
    echo "✅ Task completed successfully"
    echo "$RESPONSE" | python -m json.tool
else
    echo "⚠️  Task status: $(echo "$RESPONSE" | grep -o '"status":"[^"]*"')"
    echo "Full response: $RESPONSE"
fi
```

Step 19: Submit Multiple Task Types

bash

```
# Test 1: Spot Migration Task
curl -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "task_type": "migrate_to_spot",
    "agent_type": "cost",
    "parameters": {
      "instance_ids": ["i-123", "i-456", "i-789"]
    }
  }'

# Wait a moment
sleep 3

# Test 2: Right-sizing Task
curl -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "task_type": "right_size",
    "agent_type": "cost",
    "parameters": {
      "instance_ids": ["i-abc", "i-def"]
    },
    "priority": 10
  }'

# Expected: Both tasks should be submitted and processed
```

Step 20: List All Tasks

bash

```
# List all tasks
curl http://localhost:8080/tasks

# Expected output:
# {
#   "tasks": [
#     {
#       "task_id": "...",
#       "task_type": "analyze_cost",
#       "status": "completed",
#       ...
#     },
#     {
#       "task_id": "...",
#       "task_type": "migrate_to_spot",
#       "status": "completed",
#       ...
#     },
#     {
#       "task_id": "...",
#       "task_type": "right_size",
#       "status": "completed",
#       ...
#     }
#   ],
#   "count": 3
# }
```

Step 21: Filter Tasks by Status

bash

```
# Get only completed tasks
curl "http://localhost:8080/tasks?status=completed"

# Get only pending tasks (should be empty)
curl "http://localhost:8080/tasks?status=pending"

# Get failed tasks (should be empty if all worked)
curl "http://localhost:8080/tasks?status=failed"
```

Step 22: Test Task Timeout

```bash
# Stop the cost agent (Ctrl+C in agent terminal)
# Then submit a task

curl -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "task_type": "analyze_cost",
    "agent_type": "cost",
    "parameters": {
      "account_id": "timeout-test"
    },
    "timeout_seconds": 5
  }'

# Save task ID and check status after 10 seconds
sleep 10

curl http://localhost:8080/tasks/TASK_ID

# Expected: status should be "failed" or "timeout"
# Then restart the cost agent for remaining tests
```

Step 23: Test Task Retry Logic

Create a test script to verify retry behavior:

```bash
```

```
cd ~/optiinfra

cat > test_task_retry.py << 'EOF'
#!/usr/bin/env python3
"""Test task retry logic."""

import requests
import time
import json

ORCHESTRATOR_URL = "http://localhost:8080"

def submit_task():
    """Submit a task."""
    response = requests.post(
        f"{ORCHESTRATOR_URL}/tasks",
        json={
            "task_type": "analyze_cost",
            "agent_type": "cost",
            "parameters": {"account_id": "retry-test"},
            "timeout_seconds": 10,
            "max_retries": 3
        }
    )
    return response.json()

def check_task(task_id):
    """Check task status."""
    response = requests.get(f"{ORCHESTRATOR_URL}/tasks/{task_id}")
    return response.json()

def main():
    print("=== Testing Task Retry Logic ===\n")

    # Submit task
    print("1. Submitting task...")
    result = submit_task()
    task_id = result['task_id']
    print(f"   Task ID: {task_id}")
    print(f"   Status: {result['status']}\n")

    # Poll for completion
    print("2. Polling task status...")
```

```
    for i in range(30):
        time.sleep(1)
        status = check_task(task_id)
        print(f"   [{i+1}s] Status: {status['status']}, Retries: {status['retry_count']}")

        if status['status'] in ['completed', 'failed', 'timeout']:
            print(f"\n3. Final status: {status['status']}")
            if status.get('result'):
                print(f"   Result: {json.dumps(status['result'], indent=2)}")
            if status.get('error'):
                print(f"   Error: {status['error']}")
            break

    print("\n=== Test Complete ===")

if __name__ == "__main__":
    main()
EOF

chmod +x test_task_retry.py
python test_task_retry.py
```

Step 24: Test Agent Health Monitoring

bash

```
# Stop the cost agent (Ctrl+C in agent terminal)

# Wait 45 seconds for health check to mark it unreachable
sleep 45

# Check agent status
curl http://localhost:8080/agents

# Expected: agent status should be "unreachable"

# Restart the cost agent
cd ~/optiinfra/agents/cost_agent
export PYTHONPATH="$HOME/optiinfra:$PYTHONPATH"
python main.py &

# Wait a few seconds and check again
sleep 5
curl http://localhost:8080/agents

# Expected: agent status should be "healthy" again
```

✅ COMPREHENSIVE VERIFICATION

Run this complete verification script:

```
bash
```

```
cd ~/optiinfra

cat > verify_task_routing.sh << 'EOF'
#!/bin/bash

echo "================================================================================"
echo "FOUNDATION-0.7 COMPREHENSIVE VERIFICATION"
echo "================================================================================"

# Colors
GREEN='\033[0;32m'
RED='\033[0;31m'
YELLOW='\033[1;33m'
NC='\033[0m' # No Color

# Test counter
PASSED=0
FAILED=0

# Test function
test() {
    if [ $? -eq 0 ]; then
        echo -e "${GREEN} ✅ $1${NC}"
        ((PASSED++))
    else
        echo -e "${RED} ❌ $1${NC}"
        ((FAILED++))
    fi
}

echo ""
echo "1. ORCHESTRATOR HEALTH"
curl -s http://localhost:8080/health | grep -q "healthy"
test "Orchestrator is healthy"

echo ""
echo "2. AGENT REGISTRATION"
AGENT_COUNT=$(curl -s http://localhost:8080/agents | grep -o '"count":[0-9]*' | grep -o '[0-9]*')
if [ "$AGENT_COUNT" -gt 0 ]; then
    echo -e "${GREEN} ✅ Found $AGENT_COUNT registered agent(s)${NC}"
    ((PASSED++))
else
    echo -e "${RED} ❌ No agents registered${NC}"
```

```
    ((FAILED++))
    echo "Please start the cost agent first!"
    exit 1
fi

echo ""
echo "3. TASK SUBMISSION"
RESPONSE=$(curl -s -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "task_type": "analyze_cost",
    "agent_type": "cost",
    "parameters": {"account_id": "verify-test", "period": "last_7_days"}
  }')

TASK_ID=$(echo "$RESPONSE" | grep -o '"task_id":"[^"]*"' | cut -d'"' -f4)

if [ -n "$TASK_ID" ]; then
    echo -e "${GREEN} ✅ Task submitted: $TASK_ID${NC}"
    ((PASSED++))
else
    echo -e "${RED} ❌ Task submission failed${NC}"
    echo "Response: $RESPONSE"
    ((FAILED++))
    exit 1
fi

echo ""
echo "4. TASK EXECUTION"
echo -e "${YELLOW}    Waiting for task to complete (max 10s)...${NC}"

for i in {1..10}; do
    sleep 1
    STATUS=$(curl -s http://localhost:8080/tasks/$TASK_ID | grep -o '"status":"[^"]*"' | cut -d'"' -f4)
    echo "    [$i/10] Status: $STATUS"

    if [ "$STATUS" = "completed" ]; then
        break
    fi
done

curl -s http://localhost:8080/tasks/$TASK_ID | grep -q '"status":"completed"'
test "Task completed successfully"
```

```
echo ""
echo "5. TASK RESULT VALIDATION"
RESULT=$(curl -s http://localhost:8080/tasks/$TASK_ID)

echo "$RESULT" | grep -q '"result"'
test "Task has result field"

echo "$RESULT" | grep -q '"account_id":"verify-test"'
test "Result contains correct account_id"

echo "$RESULT" | grep -q '"potential_savings"'
test "Result contains savings data"

echo ""
echo "6. MULTIPLE TASK TYPES"

# Submit spot migration task
SPOT_TASK=$(curl -s -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "task_type": "migrate_to_spot",
    "agent_type": "cost",
    "parameters": {"instance_ids": ["i-test1", "i-test2"]}
  }' | grep -o '"task_id":"[^"]*"' | cut -d'"' -f4)

sleep 3

curl -s http://localhost:8080/tasks/$SPOT_TASK | grep -q '"status":"completed"'
test "Spot migration task completed"

# Submit right-sizing task
RIGHTSIZE_TASK=$(curl -s -X POST http://localhost:8080/tasks \
  -H "Content-Type: application/json" \
  -d '{
    "task_type": "right_size",
    "agent_type": "cost",
    "parameters": {"instance_ids": ["i-test3"]}
  }' | grep -o '"task_id":"[^"]*"' | cut -d'"' -f4)

sleep 2

curl -s http://localhost:8080/tasks/$RIGHTSIZE_TASK | grep -q '"status":"completed"'
test "Right-sizing task completed"
```

```
echo ""
echo "7. TASK LISTING"
TASK_LIST=$(curl -s http://localhost:8080/tasks)

echo "$TASK_LIST" | grep -q "$TASK_ID"
test "Original task appears in list"

TASK_COUNT=$(echo "$TASK_LIST" | grep -o '"count":[0-9]*' | grep -o '[0-9]*')
if [ "$TASK_COUNT" -ge 3 ]; then
    echo -e "${GREEN} ✅ Found $TASK_COUNT tasks in list${NC}"
    ((PASSED++))
else
    echo -e "${RED} ❌ Expected at least 3 tasks, found $TASK_COUNT${NC}"
    ((FAILED++))
fi

echo ""
echo "8. TASK FILTERING"
curl -s "http://localhost:8080/tasks?status=completed" | grep -q '"count"'
test "Can filter tasks by status"

echo ""
echo "================================================================================"
echo "VERIFICATION SUMMARY"
echo "================================================================================"
echo -e "Passed: ${GREEN}$PASSED${NC}"
echo -e "Failed: ${RED}$FAILED${NC}"

if [ $FAILED -eq 0 ]; then
    echo -e "\n${GREEN} ✅ ALL TESTS PASSED!${NC}"
    echo "Task routing system is fully operational!"
    exit 0
else
    echo -e "\n${RED} ❌ SOME TESTS FAILED${NC}"
    exit 1
fi
EOF

chmod +x verify_task_routing.sh
./verify_task_routing.sh
```

Expected Output:

```
================================================================================
FOUNDATION-0.7 COMPREHENSIVE VERIFICATION
================================================================================

1. ORCHESTRATOR HEALTH
✅ Orchestrator is healthy

2. AGENT REGISTRATION
✅ Found 1 registered agent(s)

3. TASK SUBMISSION
✅ Task submitted: abc12345-...

4. TASK EXECUTION
   Waiting for task to complete (max 10s)...
   [1/10] Status: sent
   [2/10] Status: running
   [3/10] Status: completed
✅ Task completed successfully

5. TASK RESULT VALIDATION
✅ Task has result field
✅ Result contains correct account_id
✅ Result contains savings data

6. MULTIPLE TASK TYPES
✅ Spot migration task completed
✅ Right-sizing task completed

7. TASK LISTING
✅ Original task appears in list
✅ Found 3 tasks in list

8. TASK FILTERING
✅ Can filter tasks by status

================================================================================
VERIFICATION SUMMARY
================================================================================

Passed: 13
Failed: 0
```

✅ ALL TESTS PASSED!

Task routing system is fully operational!

🐛 TROUBLESHOOTING

Issue 1: Build Fails - Import Cycle

Symptoms:

```
import cycle not allowed
package optiinfra/services/orchestrator/internal/task
```

Solution:

```bash
cd ~/optiinfra/services/orchestrator
go mod tidy
go clean -cache
go build -o bin/orchestrator ./cmd/server
```

Issue 2: Agent Can't Connect to Orchestrator

Symptoms:

```
Registration failed: Connection refused
```

Solution:

```bash
```

```
# Check if orchestrator is running
curl http://localhost:8080/health

# If not running, start it
cd ~/optiinfra/services/orchestrator
./bin/orchestrator

# Verify Redis is running
docker ps | grep redis
```

Issue 3: Task Submission Fails - No Agent Available

Symptoms:

```
{"error": "no available agent: no healthy agents available"}
```

Solution:

```
bash

# Check if agent is registered and healthy
curl http://localhost:8080/agents

# If no agents, start the cost agent
cd ~/optiinfra/agents/cost_agent
export PYTHONPATH="$HOME/optiinfra:$PYTHONPATH"
python main.py

# Wait a few seconds and retry
```

Issue 4: Task Hangs in "sent" Status

Symptoms: Task status stays "sent" and never completes.

Solution:

```
bash
```

```
# Check agent logs for errors
# Agent terminal should show:
# "Received task: task-id (type: analyze_cost)"
# "Task completed: task-id (1500ms)"

# If agent crashed, restart it
cd ~/optiinfra/agents/cost_agent
python main.py

# Check if agent endpoint is reachable
curl http://localhost:8001/health
```

Issue 5: Python Import Error

Symptoms:

```
ModuleNotFoundError: No module named 'shared'
```

Solution:

```
bash

# Set PYTHONPATH correctly
export PYTHONPATH="$HOME/optiinfra:$PYTHONPATH"

# Or add to ~/.bashrc
echo 'export PYTHONPATH="$HOME/optiinfra:$PYTHONPATH"' >> ~/.bashrc
source ~/.bashrc

# Retry starting agent
cd ~/optiinfra/agents/cost_agent
python main.py
```

Issue 6: Flask Module Not Found

Symptoms:

```
ModuleNotFoundError: No module named 'flask'
```

Solution:

```bash
# Install Flask
pip install flask requests

# Verify installation
python -c "import flask; print('Flask installed')"
```

📊 WHAT YOU HAVE NOW

Complete Task Routing System:

```
OptiInfra Architecture:
├── Databases ✅
│   ├── PostgreSQL (18 tables)
│   ├── ClickHouse (8 tables)
│   ├── Qdrant (3 collections)
│   └── Redis (registry + tasks)
│
└── Orchestrator (Go) ✅
    ├── HTTP Server (Gin)
    ├── Health endpoint
    │
    ├── Agent Registry ✅ (0.6)
    │   ├── Registration API
    │   ├── Heartbeat monitoring
    │   ├── Discovery APIs
    │   └── Health checker
    │
    └── Task Router ✅ NEW (0.7)
        ├── Task submission API
        ├── Task status tracking
        ├── HTTP client to agents
        ├── Retry logic
        ├── Load balancing (basic)
        └── Redis-backed queue

Total Code: ~2,500 lines (Go + Python)
```

Capabilities Unlocked:

✅ **Task Submission**

- Submit tasks via REST API
- Route to appropriate agent type
- Priority handling
- Timeout configuration
- Max retry configuration

✅ **Task Execution**

- Automatic agent selection
- HTTP POST to agent endpoints
- Async execution
- Result collection
- Error handling

✅ **Task Monitoring**

- Real-time status tracking
- List all tasks
- Filter by status
- Get task results
- Cancel pending tasks

✅ **Reliability**

- Automatic retries on failure
- Timeout handling
- Agent health checking
- Graceful degradation

✅ **Agent Integration**

- Easy Python SDK

- Flask-based task handler
- Automatic task routing
- Response handling

🎊 MILESTONE ACHIEVED

✅ FOUNDATION-0.7 COMPLETE!

You now have:

- ✅ Complete task routing system in Go
- ✅ 4 REST APIs for task management
- ✅ HTTP client with retry logic
- ✅ Redis-backed task persistence
- ✅ Python SDK for agents
- ✅ Complete working agent example
- ✅ Comprehensive testing

Foundation Phase Progress:

```
Week 1 Progress: 9/15 prompts (60%)
├── 0.2a: Core Schema ✅
├── 0.2b: Agent State ✅
├── 0.2c: Workflow History ✅
├── 0.2d: Resource Schema ✅
├── 0.2e: Analytics Schema ✅
├── 0.3: ClickHouse ✅
├── 0.4: Qdrant ✅
├── 0.6: Agent Registry ✅
└── 0.7: Task Routing ✅ NEW

Remaining: 6 prompts (40%)
```

🎯 WHAT'S NEXT

FOUNDATION-0.8: Workflow Engine (CRITICAL - Next Priority)

- Define multi-step workflows
- Chain agent tasks together
- Handle dependencies
- Rollback on failure

Example Workflow:

```
Cost Optimization Workflow:
1. Cost Agent: Analyze current spend
2. Cost Agent: Generate recommendations
3. Application Agent: Validate quality impact
4. Cost Agent: Execute approved changes
5. Performance Agent: Monitor impact
```

📝 NOTES FOR NEXT PHASE

Ready for Workflow Engine:

- ✅ Agents can register
- ✅ Tasks can be routed
- ✅ Results are tracked
- ✅ Failures are handled

What Workflow Engine Needs:

- Workflow definition format
- Step execution logic
- Dependency management
- State persistence
- Conditional logic
- Parallel execution

Integration Points:

- Use task router for execution
- Use registry for agent discovery

- Store workflow state in PostgreSQL
- Track execution in ClickHouse

🎉 CONGRATULATIONS!

You've built a production-ready task routing system that enables:

- Orchestrator to agent communication
- Automatic agent selection
- Reliable task execution
- Comprehensive error handling
- Easy agent development

Your OptiInfra platform can now:

1. Register multiple agents
2. Route tasks to appropriate agents
3. Execute tasks with retries
4. Track task status and results
5. Handle agent failures gracefully

Next step: Build the workflow engine to orchestrate complex multi-agent operations! 🚀